Progress Report

An Automatic Parallelization Tool For Sequential Programs

Jon Flower, Adam Kolawa
ParaSoft Corporation
2500 E. Foothill Blvd., Suite 205
Pasadena, CA 91107

(818) 792-9941

Contract: N00014-90-C-0141
Topic Number SDIO-90-010
October 25, 1990

An Automatic Parallelization Tool for Sequential Programs[1]

An Interim Report

J.W.Flower[2], A.Kolawa

ParaSoft Corporation
2500, E. Foothill Blvd.
Pasadena, CA 91107

Phone: (818)-792-9941
FAX: (818)-792-0819

1. Research sponsored by SDIO/IST and managed by ONR/NOSC under the SBIR program.
2. Principal Investigator

1. Overview

In this project we have been trying to improve a system called *ASPAR* - the "automatic, symbolic parallelization" system. This is a software tool designed to automate the process of converting sequential C and Fortran programs for execution on parallel computers.

Our strategy, as developed in our own internal research is to modify sequential programs by the addition of calls to the *Express* runtime library. This is a set of library utilities specifically designed to allow parallel programs to operate in a manner independent of the underlying hardware. A set of abstractions is provided in which the physical machine topology can be hidden by a programming interface in which the machine interconnections can be assumed to match the topology of the data being manipulated. This ability is of central importance to our automated techniques: multi-dimensional arrays can be mapped logically onto a similarly multi-dimensional parallel processing network regardless of that machine physical interconnectivity.

A second important advantage of *Express* is its portability. *Express* programs can be executed on a wide range of different parallel processing systems including nCUBE and INTEL hypercubes, multi-headed Crays, transputer arrays and networks of UNIX workstations. This allows us to test our methods on many different parallel architectures rather than concentrating on a single type of system. As a result our tools will be of wide relevance in the parallel processing community.

The preliminary version of *ASPAR* was able to generate parallel C programs from a standard ANSI C program contained in a single source file. Our goals in the Phase I proposal were basically to take this prototype and convert it into a beta-release product capable of dealing with large scale applications written in C. Some efforts would also be directed towards an analysis of the techniques required to implement Fortran and even Ada versions of the tool.

2. Completed work, work in progress

Several stages of the preliminary work plan have been completed or are in progress as described in the following sections.

2.1 *ASPAR* Evaluation.

Our first goal in improving *ASPAR* was to evaluate its performance on typical algorithms. By this method it was hoped that the strengths and weaknesses of the existing system could be analyzed and any necessary modifications designed and implemented. To perform this evaluation we analyzed several algorithms in-house and also sent alpha-release copies of *ASPAR* to some of our developers.

In general the results were very encouraging. Despite obvious weaknesses in several areas everyone was uniformly delighted by the abilities of *ASPAR* to perform both simple and complex operations on their algorithms.

At its simplest *ASPAR* was able to remove the tedious "loop range and array index modifications" that plague the parallelization of simple grid based applications. This type of work is fairly straightforward but extremely error prone and the ability of *ASPAR* to perform it automatically is a great success.

In more complex areas *ASPAR* was able to parallelize quite complex algorithms by inserting appropriate calls to *Express* routines to redistribute data. Even in cases where no obvious grid based decomposition could be identified *ASPAR* was able to create parallel algorithms by continually moving data to match distributed loop ranges.

2.1.1 An Example: Conjugate Gradient Matrix Inversion.

As an example of the types of operations performed by *ASPAR* consider the problem of banded--matrix inversion by the Conjugate Gradient algorithm. This problem has been extensively studied in the parallel computing literature due to its central role in many applications. While the basic algorithms are well understood the parallelization of a "banded" solver is by no means straightforward. Standard partitioning strategies, which work well for full matrices, must be modified to take into account the banded nature. Obviously one solution is to fill in all the zeros and use a full matrix solver but this is unacceptable.

The original C source code is shown in Figure 1.

```
/****** Solver for linear equations by CG method ******/
#include "stdio.h"
#define SZ 400
#define BND 100
#define ep (double)1.0e-14
double eps;
double ar[SZ][BND],bb[SZ],xx[SZ],P[SZ],X[SZ];
double R[SZ],newR[SZ],newP[SZ],KP[SZ],newX[SZ];

void band_multi(A,x,b,elm,hbnd)
{
   int elm,hbnd;
   double A[SZ][BND],x[],b[];
{
   int i,j,band;
   double sum;
   band=hbnd*2-1;
   for(i=0;i<elm;i++){
        sum=0.0;
        if(i<hbnd)
             for(j=0;j<band;j++)sum+=A[i][j]*x[j];
        else
             for(j=0;j<band;j++)sum+=
                  A[i][j]*x[(j+1+(i-hbnd))];
        b[i]=sum;
   }
}
main()
{
   int i,j,k,elm,hbnd,param[2];
   double sum1,sum2,rd,alpha,beta;
   read_data(ar,xx,bb,param);
   elm=param[0],hbnd=param[1];
   for(i=0;i<elm;i++){
        X[i]=0.0;
        P[i]=bb[i];
        R[i]=bb[i];
   }
   sum1=0.0;
   for(i=0;i<elm;i++)
        sum1+=bb[i]*bb[i];
   eps=ep*sum1;
   k=1,rd=(double)100000.0;
   while(rd>eps){
        band_multi(ar,P,KP,elm,hbnd);
        sum1=sum2=0.0;
        for(i=0;i<elm;i++)sum1+=R[i]*R[i];
        for(i=0;i<elm;i++)sum2+=P[i]*KP[i];
             alpha=sum1/sum2;
        for(i=0;i<elm;i++)newX[i]=X[i]+alpha*P[i];
        for(i=0;i<elm;i++)newR[i]=R[i]-alpha*KP[i];
        sum2=0.0;
        for(i=0;i<elm;i++)sum2+=newR[i]*newR[i];
        beta=sum2/sum1;
        for(i=0;i<elm;i++)
        newP[i]=newR[i]+beta*P[i];
        rd=sum1;
        for(i=0;i<elm;i++){
             P[i]=newP[i];
             R[i]=newR[i];
             X[i]=newX[i];
        }
        k++;
   }
   print_result(X);
}
```

Figure 1 Sequential Program for Conjugate Gradient Matrix Inversion

The basic algorithm is fairly straightforward, all of the code is shown except for the I/O routines: `read_data` and `print_result`.

The results of applying *ASPAR* to this code are shown in Figure 2.

```
*$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$*/
/*$  AUTOMATICALLY PARALLELIZED PROGRAM     $*/
/*$  Decompose this as 1-D problem          $*/
/*$  P1: proc.number of 1st dimension       $*/
/*$  AS_procs:Number of processors          $*/
/*$  AS_lgc:Logical processor location (logical no)  $*/
/*$  AS_lst:Table of logical-physical proc.numb      $*/
/*$  AS_ofst:Offset for undecomposed data   $*/
/*$  AS_cnt:No.of iteration after parallelized       $*/
/*$$$$$$$$$$$----EXPRESS header----$$$$$$$$$$$$$$$$$*/
#define P1(1)   /*-You may change it--*/
#include <express.h>
struct nodenv env;
int AS_lgc[3],AS_procs[3],AS_lst[100],
    AS_type=123,AS_ofst[3],AS_cnt[3],AS_size[3];
/*******Solver for linear equations by C G method *******/
#include "stdio.h"
#define SZ 400
#define BND 100
#define ep (double)1.0e-14
double eps;
double ar[SZ/P1][BND],bb[SZ/P1],xx[SZ/P1],P[SZ],X[SZ/P1];
double R[SZ/P1],newR[SZ/P1],newP[SZ/P1],KP[SZ/P1],
    newX[SZ/P1];
void band_multi(A,x,b,elm,hbnd)
    int elm,hbnd;
    double A[SZ/P1][BND],x[],b[];
{
    int i,j,band;
    double sum;
    band=hbnd*2-1;
/*$ PARALLEL $*/
    AS_set_range(0,elm-1,0,1);
    for(i=0;i<AS_cnt[0];i++){
        sum=0.0;
    if(i+AS_ofst[0]<hbnd)
/*$ Sequential: Data-Decomp Strategy2 $*/
    for(j=0;j<band;j++)
        sum+=A[i][j]*x[j];
    else
/*$ Sequential: Data-Decomp Strategy2 $*/
        for(j=0;j<band;j++)
            sum+=A[i][j]*x[(j+1+(i+AS_ofst[0]-hbnd))];
        b[i]=sum;
    }
}
main()
{
    int i,j,k,elm,hbnd,param[2];
    double sum1,sum2,rd,alpha,beta;
/*$$$$$$$$ Initialization for EXPRESS $$$$$$$$$*/
    exparam(&env);
    AS_procs[0]=env.nprocs;
    exgridinit(1,AS_procs);
    exgridcoord(env.procnum,AS_lgc);
    exconcat(AS_lgc,4,AS_lst,4,NULLPTR,ALLNODES,
            NULLPTR,&AS_type);
/*$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$*/
    read_data(ar,xx,bb,param);
    elm=param[0],hbnd=param[1];
/*$ PARALLEL $*/
    AS_set_range(0,elm-1,0,1);
    for(i=0;i<AS_cnt[0];i++){
        X[i]=0.0;
        P[i+AS_ofst[0]]=bb[i];
        R[i]=bb[i];
    }
/*$ CONCATENATEP $*/
    AS_size[0]=sizeof(*P)*AS_num[0];
    exconcat(&P[AS_ofst[0]],AS_size[0],P,AS_size[0],
            NULLPTR,env.nprocs,AS_lst,&AS_type);
    sum1=0.0;
/*$ PARALLEL $*/
    AS_set_range(0,elm-1,0,1);
    for(i=0;i<AS_cnt[0];i++)
        sum1+=bb[i]*bb[i];
/*$ COMBINE sum1 by PLUS $*/
    excombine(&sum1,f_add,sizeof(sum1),1,ALLNODES,
            NULLPTR,&AS_type);
    eps=ep*sum1;
    k=1,rd=(double)100000.0;
    while(rd>eps){
        band_multi(ar,P,KP,elm,hbnd);
        sum1=sum2=0.0;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)sum1+=R[i]*R[i];
/*$ COMBINE sum1 by PLUS $*/
        excombine(&sum1,f_add,sizeof(sum1),1,ALLNODES,
                NULLPTR,&AS_type);
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)
            sum2+=P[i+AS_ofst[0]]*KP[i];
/*$ COMBINE sum2 by PLUS $*/
        excombine(&sum2,f_add,sizeof(sum2),1,ALLNODES,
                NULLPTR,&AS_type);
        alpha=sum1/sum2;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)
            newX[i]=X[i]+alpha*P[i+AS_ofst[0]];
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)newR[i]=R[i]-alpha*KP[i];
        sum2=0.0;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)sum2+=newR[i]*newR[i];
/*$ COMBINE sum2 by PLUS $*/
        excombine(&sum2,f_add,sizeof(sum2),1,ALLNODES,
                NULLPTR,&AS_type);
        beta=sum2/sum1;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++)
            newP[i]=newR[i]+beta*P[i+AS_ofst[0]];
        rd=sum1;
/*$ PARALLEL $*/
        AS_set_range(0,elm-1,0,1);
        for(i=0;i<AS_cnt[0];i++){
            P[i+AS_ofst[0]]=newP[i];
            R[i]=newR[i];
            X[i]=newX[i];
        }
/*$ CONCATENATE P $*/
        AS_size[0]=sizeof(*P)*AS_num[0];
        exconcat(&P[AS_ofst[0]],AS_size[0],P,AS_size[0],
                NULLPTR,env.nprocs,AS_lst,&AS_type).
        k++;
    }
    print_result(X);
}
```

Figure 2 Conjugate Gradient code parallelized by *ASPAR*

2.1.2 Successes of the Existing System.

The most dramatic success of *ASPAR* is that the program shown in Figure 2 is a correct, fully parallel conjugate gradient matrix inverter. All aspects of the displayed code, including the comments, were automatically inserted by the system.

The performance of the method is summarized by the data shown in Figure 3. The data shown was

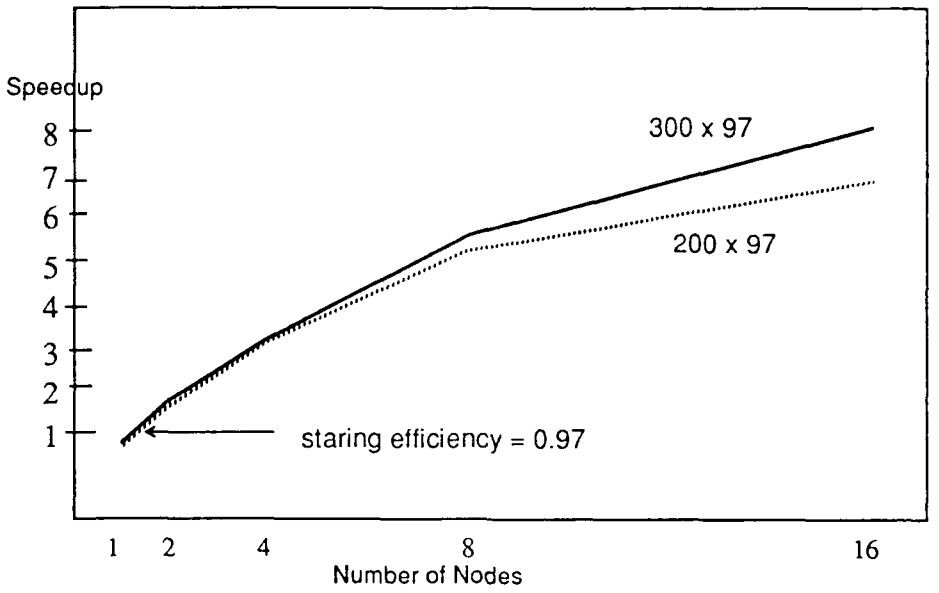

Figure 3 Performance of automatically parallelized matrix inverter

The performance of the Conjugate Gradient is shown for matrices of order 200 and 300 with bandwidth 97. Data collected from an nCUBE/10

collected by running the program for matrices of order 200 and 300 with bandwidth 97 in both cases. The initial data point, for a single processor, shows the comparison between the original sequential program running on a single node and the new parallel program running on a single node.

These results are extremely good. Despite the somewhat irregular nature of the sequential algorithm *ASPAR* was able to successfully generate a parallel code whose efficiency is greater than 50% on up to 16 nodes.

2.1.3 Problems with the Current System.

Perhaps the most obvious defect with the code shown in Figure 2 is its illegibility. Despite the insertion of automated comments which describe the parallelization strategies being implemented the resulting C code is messy and hard to read. The *Express* system calls have many arguments whose relationship to the underlying decomposition schemes is obvious only to those intimately familiar with the system.

While this is not necessarily a bad feature in cases such as this where parallelization has been accomplished successfully it is a problem in those cases where only partial success is achieved. In

these cases we must be able to make available to the user enough information to complete the process successfully and this is hindered if the generated code is too complex.

A second, and more serious problem surrounds the routines omitted from Figure 2: `read_data` and `print_result`.

Originally it was intended that *ASPAR* would use the parallel I/O constructs found in the *Express* runtime library. In common with other utilities these routines are tailored to match the underlying abstraction of machine architecture. For example, one can ask to read an array and simultaneously distribute it over a two-dimensional processor decomposition. The problems in attempting this type of solution for *ASPAR* are

- Generalizing the I/O modes for arbitrary decomposition strategies, including higher-dimensional grids is difficult.
- Support for "parallel" high speed I/O devices is difficult.
- Any "complex" I/O statement in the original sequential program is impossible to parallelize.

This latter problem is the most serious. Simple format I/O can, in principle, be dealt with although this requires solutions to the first two problems indicated above.

The last problem is extremely difficult to solve. A simple sequential file containing an array of integer values is easy enough. Consider, however, an I/O statement of the following type

```
int i, j;
float A[10], B[10][20];

fscanf(fp, "%d %d %f %f\n", &i, &j, &A[i], &B[i][j]);
```

This code is obviously quite complex especially when one adds the fact that the arrays `A` and `B` are distributed among the processors, quite possibly in totally different ways.

A simple solution to this type of difficulty is to disallow it in any program presented to *ASPAR*. Since this type of construct is actually reasonably rare in large scale C programs this might be acceptable. Fortran, however, often contains this type of code.

Another cause of problems with programs presented to *ASPAR* was caused by the "static" nature of the analysis performed. This problem is similar to that faced by most compiler technologies when the exact runtime behavior of a code cannot be determined at compile time because of data dependencies.Analogously we found that *ASPAR* could only successfully parallelize programs whose runtime data decompositions were known at compile time. This problem was caused by an apparent mismatch between *ASPAR*'s knowledge of the underlying structure of an algorithm and the abilities of the *Express* runtime system to support those ideas. While most combinations could, in principle, be created the work involved was often prohibitive.

A final problem concerned programs whose data decomposition requirements varied during the course of the application. The original *ASPAR* had no way to track data distribution dynamically and thus had to periodically transform an existing data decomposition to some standard form in order to proceed.

2.2 Designing a better *ASPAR.*

Because of the weaknesses mentioned above we decided to make the following general improvements to the *ASPAR* system:

- Support for programs built from multiple source files.
- A more sophisticated I/O subsystem.
- Alternative interfaces to the *Express* runtime system to provide more and easier to understand support.

2.2.1 Support for Multiple Source Files

The simplest improvement required is to provide support for programs consisting of more than one source file. This requires implementation of an *ASPAR* "linker" which builds a data-base containing information about the program being parallelized. The information in this data-base is then used by a second *ASPAR* pass which performs the linking operations to resolve information contained in separate files and produce an overall parallelization strategy.

Once the overall decomposition strategy has been decided each of the individual source files which makes up the program is converted to parallel form.

This tool has been successfully completed.

2.2.2 I/O Interface

The best solution to the I/O problems discussed above is a "remote procedure call" mechanism in which sections of the original source code are executed, *in tact*, on some processor which has access to the physical medium on which the data is stored.

Our original idea was to use the advanced I/O modes of *Cubix* to accomplish parallel I/O as shown schematically in Figure 4. The call to `fscanf`, for example, in the parallel program running on

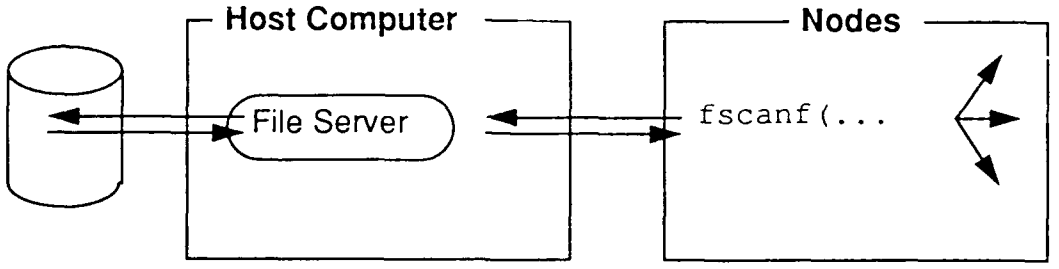

Figure 4 I/O to a parallel program using *Cubix* I/O modes

the nodes generates a request to "read data" in the file server process running on the node attached

to the physical disk. This process reads a certain amount of data, without processing it, and returns it to the calling node which then processes it according to the format specifier finally distributing it to other nodes according to the *Cubix* I/O mode. The problem with this mechanism is that all the intelligence resides in the parallel computer node. This means that to use *ASPAR* effectively would require that we deduce all possible file formats *at compile time* - a hopeless task. Note that there are already problems with a single call to `fscanf`. If this were an element in a more complex I/O procedure the problem would be even more complex.

Our solution to this problem is to replace the I/O call in the node program with a routine which invokes a procedure on the file serving node. This procedure is generated automatically by *ASPAR* by taking the I/O statements from the original sequential program and encapsulating them in an *Express* protocol harness. The overall picture is rather more like that of Figure 5.

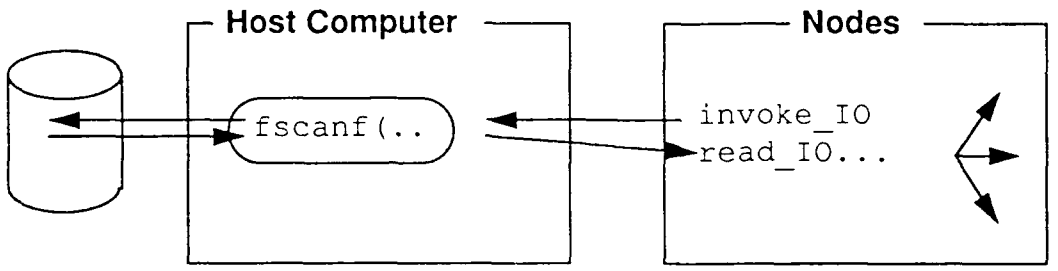

Figure 5 Parallel I/O generated as a remote procedure call

The call to `fscanf` in the nodes has been replaced by something which merely causes the original sequential code to be executed on the file server. This call obviously interprets the file format in exactly the same way as the original sequential program did. Once the data has been converted into the internal memory of the file server it is sent back to the calling node where a second procedure call reads and distributes the incoming data.

This method has several important advantages

- All the internal file format processing is carried out on the file server which is (presumably) similar in capability to the machine on which the original sequential program ran. As a result there are no ambiguities about which element of a distributed array get which data - there are no distributed arrays in the fileserver node.
- The multiplexing of data required when redistributing the data into the individual memories of the node processors is identical in function to the internal data redistributions required by other *ASPAR* operations although the original source of data is the file server rather than another node processor. Since *Express* treats the file serving node as just another node this difference is transparent.

- The call to `read_IO` need not necessarily follow immediately after the call to `invoke_IO`. In particular the `invoke_IO` call may be moved to an earlier "time" in the programs execution to allow the disk I/O to be overlapped with computation in the node processors.
- The mapping of remote procedures to file servers can be made dynamic allowing the use of either a conventional file system or some optimized parallel I/O resource.
- The I/O statements can take any form supported by the file server's compilers. This is often important in leading edge parallel processing systems where compiler technology often lags somewhat behind workstations and other file serving machines.

We are currently creating a functional specification for this I/O interface and will complete a preliminary implementation within Phase I of this project.

2.2.3 An alternative *Express* Interface

In most applications, especially those in which *ASPAR* was able to successfully parallelize the input program the *Express* message passing primitives were found to be adequate. In examining the failures, however, we have identified the following two issues:

- The interface must be made "dynamic" in the sense that *ASPAR* should be allowed to request any decomposition at any time without regard for where the data might currently be located.
- The interface should be simplified so that the end user can comprehend the communication strategy implemented or partially implemented by *ASPAR*.

To achieve these two goals we have designed a new set of *Express* utilities built around the two functions: `exlayout` and `exdist`.

`exlayout` allows the programmer to "register" the current data distribution of any data object. The distribution may be global, in which case each node has a copy of the same data, local in which case nodes have independent subsets of the whole or any mixture of the two extreme cases.

`exdist` uses the information registered in the most recent call to `exlayout` to redistribute data among the parallel computing nodes according to some user-specification. Any combination of existing and required decompositions is allowed and the data movement is accomplished in a time which grows only logarithmically with the number of nodes in the system.

The combination of `exlayout` and `exdist` yields an extremely powerful and very high level interface to interprocessor communication. The particular abstraction used allows the programmer to be entirely free of conventional "message passing" ideas. Instead the system call `exdist` can be considered to be responsible for bringing the data required for the next operation to this processor *regardless of its original location in the parallel machine.* Note that this very powerful concept is equally at home on distributed memory machines in which the physical data movement is essential for correct operation of the algorithm or shared memory machines with caches in which it is essential for good performance to pre-fetch and cache data which will be used in the upcoming operations.

As well as solving many of the communication related problems with the original *ASPAR* we believe that these concepts will have extremely far reaching effects in many parallel processing arenas. Already mentioned is the "caching" problem of many advanced architectures. Other important areas include the visualization of parallel algorithms and the optimization of internal communication patterns. We will have more to say about these ideas in the next few sections.

We are currently completing the specifications for these two functions and will complete initial implementations in Phase I work.

2.3 User Interface and Visualization Tools.

In using *ASPAR* and its related tools we have identified several important areas in which interactions between human and user must be presented

- Before attempting to use *ASPAR* it is important to understand the operation of the sequential algorithm and the way in which it operates on its internal data.
- While using *ASPAR* it is necessary to be able to observe the changes being made to the user code and also to graphically visualize any problems which prevent *ASPAR* from completing its task. In particular this requires sophisticated data analysis and dependency analysis tools which can visually display the interactions between the various conflicting requirements of a particular parallelization scheme.
- After *ASPAR* has parallelized the application it is necessary to be able to visualize the parallel operations taking place. This is required in order to understand and optimize the performance of the parallel program and also to validate the ordering assumptions and dependency violations that may have been introduced in parallelization.

We are currently completing the design of five new tools which will be able to support this type of data analysis. Together they should provide a sophisticated user interface to help the programmer through the parallelization process.

2.3.1 Sequential program analysis: mapv

The `mapv` tool is provided to support visualization of the algorithmic behavior of a sequential program with regard to its data access patterns. The utility operates in two modes:

- User directed.
- *ASPAR* directed.

In the former mode the user "instruments" the sequential program by including directives to monitor accesses to particular variables or arrays. Upon completion of the sequential program a database is created which contains time-stamped markers relating to the access patterns of the application.

In the *ASPAR* directed mode the user program is instrumented automatically by *ASPAR* by analyzing the data structures upon which the automated parallelization would operate.

The analysis tool presents a "playback" of the access patterns to each independent data object allowing the user to observe the "patterns" of access and correlate them with the appropriate lines of

the original source code. The speed at which the data is played back can be controlled by the user until a suitable visualization of the algorithm has been achieved.

This utility will be used to give an idea of the locality or otherwise of a program's data access requirements. In this way the user can understand the decisions that *ASPAR* makes about which data objects to distribute among processors and which to maintain as global objects, known in every node. The interrelationships between distributed objects can also help in understanding the communication system calls required by *ASPAR*.

In cases where *ASPAR* fails to parallelize an algorithm `mapv` will help in understanding the relationships cause failure.

A preliminary implementation of `mapv` is complete. We are designing and implementing new features and expect this tool to be complete by the end of Phase I research.

2.3.2 Algorithm Structure and Dependency Analysis: ftool

During the parallelization process it is necessary for the user to be able to visualize the overall structure of the program being parallelized and the relationships between data objects which affect the performance and possibilities of a parallel algorithm.

`ftool` is designed to provide these services in a multi-window tool which will be able to display the original source code together with various types of abstracted "flow-chart" which show the overall algorithmic structure. It is expected that at least some of the available options will be compatible with standard CASE technology tools and will therefore facilitate the use of this new technology on parallel programs.

As well as showing program structure `ftool` will also be able to interactively display the results of *ASPAR*'s analysis of the code. This will include formal descriptions of the dependency analysis performed by *ASPAR* together with displays that indicate the interrelationships between various data objects that affect the success, failure or performance of an automatically parallelized program.

A possible extension to `ftool` would be an interactive dialog system which allowed the user to give *ASPAR* "hints" about strategies which may not be apparent from its static data analysis. Another possibility is to use `ftool`'s displays to indicate suitable interfaces to "canned" parallel processing libraries.

A preliminary implementation of `ftool` is complete. We are designing and implementing new features and expect a fully functional version of this tool to be complete by the end of Phase I research. Some of the advanced features mentioned in the last paragraphs of this section will be deferred until Phase II.

2.3.3 Interface Construction and Validation: HNtool

An important issue in constructing parallel programs concerns the "parallel programming model" to be used. Many machines have their own programming styles as do the various software packages that execute on them. *Express* itself supports two completely separate programming models and a whole range of variations of these two.

To help the user in building a parallel version of a large scale application a tool is required which can compare the existing code against the available programming models and give advice about the performance/maintenance payoffs involved in implementing a particular strategy. Note that this type of analysis is orthogonal to that implemented by `ftool` and *ASPAR*. These latter tools operated on the code with assumptions of a basically tightly coupled architecture. `HNtool` (tentative name only) operates in a rather more loosely coupled world in which the basic building blocks are large program fragments.

As an example consider a significant image analysis package. In such a system probably 80-90% of the total code is related to the interactive and/or batch user interface and the display and storage of image data. As such it probably uses system calls tailored to a particular display device or imaging system which will almost certainly not be available to the nodes of a parallel processing system. On the other hand most of the compute time is probably spent in the few hundred or thousand lines of code that implement the image processing algorithms. This code is usually amenable to parallel processing and could be parallelized without difficulty by *ASPAR*.

The goals of this tool are to identify and classify the various pieces of an application with respect to a suitable target processor. The various trade-offs associated with various partitioning strategies should be analyzed and displayed to the user. Once a suitable decomposition of the work-load has been developed a suitable interface between the various pieces of code should be generated automatically.

This type of activity is usually undertaken by a human being with a thorough knowledge of the underlying program. Even after a suitable decomposition has been decided, however, the task of building the interface between the various pieces is often extremely troublesome - simple but bug-ridden. It is our hope that most of this difficulty can be removed by this tool.

An important improvement in parallelization possibilities to be gained from this type of tool is that the problems associated with maintaining both a sequential and parallel version of a large scale application should be dramatically reduced since the interfaces between various "blocks" are constructed automatically and *ASPAR* performs the parallelization of the compute intensive portions of the code most of the tricky work will be accomplished automatically.

We have complete a preliminary design for this tool but do not expect to implement it until Phase II of this project.

2.3.4 Parallel Data-Flow Analysis: Datamon

An obvious extension of the `mapv` concept will allow us to visualize the flow of data in the parallelized version of any application. This utility will be used as a visual guide and documentation system for the parallel processing constructs used by *ASPAR*. In many cases it will also provide key ideas in optimizing parallel algorithms.

It is important to note that, in common with the data redistribution scheme described in 2.2.3, this tool is not "message based". There already exist several tools which can display the individual message transactions between processors. This does not, however, necessarily lead to any great understanding of the underlying parallelization methods. `Datamon` (tentative name only) will be used to display the data distribution at various points in the program - a much more intuitive scheme that observing message in transit.

One important capability will be to integrate this utility with `mapv` to allow users to observe the differences in behavior between the sequential and parallel programs.

We are currently working on a specification for this utility and also an interface between it and the closely related `mapv`. We do not expect to implement any design until Phase II.

2.3.5 Parallel Communication Optimization: Dataopt

The preliminary version of *ASPAR* that existed at the beginning of Phase I research used the standard *Express* runtime system to implement its parallelization strategies. As as been indicated we are now supplementing this with alternative, more powerful, methods.

One trade-off to be considered, however, is that of performance.

The benefit of using the *Express* primitives directly is that they are individually optimized for the hardware on which they execute and thus provide very high performance. The higher level re-distribution primitive `exdist`, while also amenable to optimization for individual architectures is, by nature, more general and thus incurs higher overheads.

It is our intention to provide a "learning" mechanism for the programs parallelized by *ASPAR* in which the parallelized application records the various changes made by the redistribution functions and allows optimization to be performed off-line as a post-processing stage. We will build a system that can match the patterns of internode communication against the standard *Express* library and suggest suitable changes which might speed-up a particular piece of code.

An important benefit of this approach is that we expect to be able to offer an "optimization service" in which user's can give us their programs output and *ASPAR* built data-base and allow us to optimize communication patterns for a particular piece of target hardware. Such a service might typically involve a two-tiered approach. In the first stage only optimizations which result in portable code might be used while the second level would allow for hardware specific but non-portable optimization.

An important side-effect for *ParaSoft* would be access to the communication patterns for a wide range of applications which would allow us to extend the library of "canned" communication utilities in useful ways.

We are currently designing and implementing the "instrumentation" package required to obtain the necessary information for this tool. We expect a preliminary version of the instrumented code to be available by the end of Phase I research but will defer implementation of the optimization processes and tools until Phase II.

2.4 Other Languages.

The pre-processing system required for parsing C programs and linking various source files together is now complete. We are currently examining the possibilities and requirements for an extension to Fortran/77.

It is hoped that a simple "front-end" will suffice to generate the data-base in a format suitable for *ASPAR* while a new "back-end" will be able to convert the input source code by the addition of Fortran calls to the *Express* runtime system.

We expect to have a preliminary Fortran interface built by the end of Phase I research.

3. Remaining Phase I work

As has been seen we have currently spent most of our efforts on the design of the various pieces of technology required to create a version of *ASPAR* which meets our needs. In the remaining months of Phase I research, however, we expect to complete the following pieces of software:

- A simple, but complete, I/O interface to *ASPAR* through the remote procedure call mechanism described in 2.2.2. No optimization for overlapped or parallel I/O systems will be performed.
- An preliminary implementation of the `exdist/exlayout` system calls together with the mechanisms for the internal monitoring and dynamic modification of data distributions. We will also attempt to perform rudimentary instrumentation of this system to allow for further development in Phase II.
- A complete version of the `mapv` utility fc camining the data access patterns of existing sequential programs. We will create a multi-window interface allowing visualization of several interrelated data structures and complete cross-referencing with the original source code.
- A complete version of the `ftool` utility for presenting the decisions and factors which affect the parallelization schemes adopted by *ASPAR*.
- A preliminary Fortran pre-processor.

Together with these "tools" we will also continue to modify the internals of *ASPAR* to reflect information derived from attempting to parallelize new programs.

4. Conclusions

Surprisingly, one of the most exciting things to come from the research performed so far has not been the growing success rate of *ASPAR* itself, which was expected, but the enormous growth of related ideas which it has produced. Many of the tools described in Section 2.3 have either been created directly as a consequence of our preliminary research or have been significantly extended because of it. The data distribution schemes described in 2.2.3 are completely new and, we believe, extremely important to programmers completely outside the range intended by our first design of *ASPAR*.

Another new and extremely important area of concern to emerge from this work is Fortran/90. Our original targeting of the C language was made so that we would be forced to solve many of the complex problems presented by such a sophisticated language. Fortran/77, by comparison is a much simpler language and although it has its own idiosyncracies we expect the benefits of using *ASPAR* to be much larger. Even more important than this, however, is the emerging standard for Fortran/90. Even though the standard has still to be finalized several vendors are already working on compilers for various features - Thinking Machine's compiler for the Connection Machine is an important example where sequential code written in Fortran/90 can be executed *in parallel* without change. We believe that this language will have a big impact on the high-performance and supercomputing world which can be significantly enhanced by the existence of tools such as *ASPAR*

which can generate portable parallel programs from Fortran/90 source. This is an area in which we would like to invest a large amount of Phase II effort.

Overall we believe that the project is currently exceeding its original goals although some of these achievements have been in areas which were not anticipated at the time of the original proposal.